Discrete Applied Mathematics 16 (1987) 101–111
North-Holland

SOME PARALLEL ALGORITHMS ON INTERVAL GRAPHS

Alan A. BERTOSSI and Maurizio A. BONUCCELLI
Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56100 Pisa, Italy

Received 11 April 1984
Revised 13 February 1986

Parallel algorithms are given for finding a maximum weighted clique, a maximum weighted independent set, a minimum clique cover, and a minimum weighted dominating set of an interval graph. Parallel algorithms are also given for finding a Hamiltonian circuit and the minimum bandwidth of a proper interval graph. The shared memory model (SMM) of parallel computers is used to obtain fast algorithms.

1. Introduction

In recent years, there has been considerable interest in devising parallel algorithms for solving various computational problems. Indeed, the advent of *very large scale integration* (VLSI) technology and the decline in the cost of the hardware have made it feasible to build economical computers composed of many thousands of processors which are capable of working together in parallel to solve a single problem.

Several different models of parallel computers have been proposed so far (e.g., see [10]). We focus here on the single instruction stream, multiple data stream (SIMD) model.

An SIMD computer consists of p *processors*, each capable of performing the standard arithmetic and logical operations. The processors are synchronized and operate under the control of a single instruction stream. An *enable/disable* mask is used to select the processors that are allowed to execute each instruction. The instruction is executed only by the enabled processors, while the remaining processors will be idle. All enabled processors execute the same instruction, maybe using different data. When an instruction is executed in parallel, each processor must be allowed to finish before the next instruction is started.

In an SIMD computer, the time required to communicate data from one processor to another one often dominates the overall computation time of an algorithm. In this paper, however, we shall deal only with the *shared memory model* (SMM), which has no communication delay at all.

In the SMM, all processors have access to a common memory. We assume that different processors can obtain the content of one memory location at the same time. Moreover, they may store information into different memory locations simultaneously, but no two processors should attempt to change the content of the same memory location at the same time.

Many parallel algorithms for solving matrix and graph problems as well as for sorting and scheduling have been based on the SMM (e.g. see [1, 4–6, 9, 10, 15]).

In this paper, we consider some combinatorial optimization problems on those graphs which represent intersection intervals of a line. Such graphs are one of the most useful discrete mathematical structures for modeling problems arising in the real world. In fact, they have found applications in archaeology, biology, psychology, traffic control, job scheduling, and storage information retrieval (e.g. see [7, 16]).

More formally, an *interval family* is a set $I = \{1, \ldots, n\}$ of intervals on the real line. An interval family is *proper* if no interval is properly contained within another. A graph is a (*proper*) *interval graph* if there is a one-to-one correspondence between the nodes of the graph and the intervals of a (proper) interval family such that two nodes of the graph are joined by an edge if and only if their corresponding intervals overlap. An interval graph is *weighted* if a real number w_i (the *weight*) is associated to each node i. In many problems, one is usually asked to find a subset S of the nodes which satisfies certain properties and optimizes $\sum_{i \in S} w_i$, in the weighted case, or $|S|$, in the unweighted one. Of course, this latter function is merely a special case of the former one, in which $w_i = 1$ for each node i.

Several sequential algorithms have been proposed for solving many combinatorial optimization problems on interval graphs (e.g. see [2, 3, 7, 8, 11]). As far as we are aware, however, the only parallel algorithms proposed up to now have been devised by Dekel and Sahni for solving the *maximum cardinality clique* problem and the *node coloring* problem (see [5], under the term *channel assignment*).

In this paper, we give parallel algorithms for solving several problems on interval graphs, such as finding a *maximum weighted clique*, a *maximum weighted independent set*, a *minimum cardinality clique cover*, and a *minimum weighted dominating set*. We also give parallel algorithms for finding a *Hamiltonian circuit* and the *minimum bandwidth* of a proper interval graph. The time complexity of the given algorithms for the above problems is either $O(\log n)$ or $O(\log^2 n)$.

In addition to analyzing the time complexity of the given algorithms, we also evaluate their *effectiveness of processor utilization* (EPU). This is defined relative to a specific problem P, the complexity of the fastest sequential algorithm known for P, and the proposed parallel algorithm A for P. Formally, $\mathrm{EPU}(A, P)$ is the ratio between:

(1) the time complexity of the fastest sequential algorithm known for P, and

(2) the number of processors needed by A times the time complexity of A.

By definition, $0 \leq \mathrm{EPU} \leq 1$, and an EPU close to 1 is considered good [5]. Unfortunately, a few parallel algorithms actually achieve this value. The parallel algorithms proposed in this paper have an EPU which is either $\Omega(\log n/n)$ or $\Omega(1/n^2)$.

2. Basic parallel procedures

Let $I = \{1, \ldots, n\}$ be an interval family such that the ith interval is equal to

$[a_i, b_i]$, $i=1,\ldots,n$. Without loss of generality, we assume that each interval contains both its endpoints and that no two intervals share a common endpoint. For the sake of simplicity, the interval graph G corresponding to the interval family I will be denoted as $G(I)$ and we shall deal directly only with the intervals instead of the nodes.

Given I, we define the *first* and *last* intervals of I, denoted as *first*(I) and *last*(I), as follows: *first*$(I)=i$ such that $b_i=\min\{b_k\}$, and *last*$(I)=j$ such that $b_j=\max\{b_k\}$.

For each interval i, we define the *rightmost overlapping* interval, denoted as *right*(i), in the following way: *right*$(i)=j$, if $b_j=\max\{b_k: a_k<b_i<b_k\}$, $=0$, otherwise. Intuitively, *right*(i) is the last interval to end after interval i among all intervals overlapping with interval i, provided it exists.

Finally, for each interval i, we define the *next nonoverlapping* interval, denoted as *next*(i), as follows: *next*$(i)=j$, if $b_j=\min\{b_k: b_i<a_k\}$, $=0$, otherwise. In words, *next*(i) is the first interval to end among all intervals which begin after the end of interval i, provided it exists.

Let us now see how *first*(I), *last*(I), *right*(i), and *next*(i) can be computed in parallel for all $i\in I$. For this purpose, consider the following problem. Given n numbers $x_1,\ldots,x_n$, find the index i^* such that $x_{i^*}=\max\{x_i\}$. It is well-known that this problem can be solved by a parallel algorithm running in $O(\log n)$ time and requiring $O(n/\log n)$ processors (e.g., see [6]).

Therefore, *last*(I) can be computed in $O(\log n)$ time simply by setting $x_i=b_i$ for all i and then computing i^*. Instead, to compute *right*(i) for a given i, it is sufficient to set $x_k=b_k$, if $a_k<b_i<b_k$, and $x_k=-\infty$, otherwise. Then compute i^* and set *right*$(i)=i^*$, if $x_{i^*}\neq-\infty$, or *right*$(i)=0$, if $x_{i^*}=-\infty$. Thus $O(\log n)$ time is needed to compute *right*(i) for a fixed i using $O(n/\log n)$ processors. Therefore, *right*(i) for all $i\in I$ can be computed in parallel also in $O(\log n)$ time, but using $O(n^2/\log n)$ processors.

A similar reasoning holds for *first*(I) and *next*(i) for all i. It is indeed sufficient to consider the problem of finding the index i^* such that $y_{i^*}=\min\{y_i\}$. In order to compute *next*(i) for a fixed i, we set $y_k=b_k$, if $a_k>b_i$, and $y_k=\infty$ otherwise. Then we compute i^* and set *next*$(i)=i^*$, if $y_{i^*}\neq\infty$, and *next*$(i)=0$, otherwise. Again, this can be done in $O(\log n)$ time using $O(n/\log n)$ processors. Hence *next*(i) for all $i\in I$ can be computed in $O(\log n)$ time, using $O(n^2/\log n)$ processors.

next(i) and *right*(i) may be interpreted as links. Let us generically denote either *next*(i) or *right*(i) with *link*(i). The *link*$(\,)$ values then define linked lists, arranged as in a forest of directed trees whose arcs are directed away from the leaves, as shown in Fig. 1. Each root j has of course *link*$(j)=0$.

Let us now consider the following problem. With each interval $i\in I$ associate a Boolean variable $\phi(i)$, initially set to 0. Given any interval j, we want to set $\phi(k)=1$ for each interval k appearing on the directed path joining j to the root of the tree having j as a leaf. This can be done by simultaneously collapsing the linked lists and transmitting the value 1 within the lists as follows:

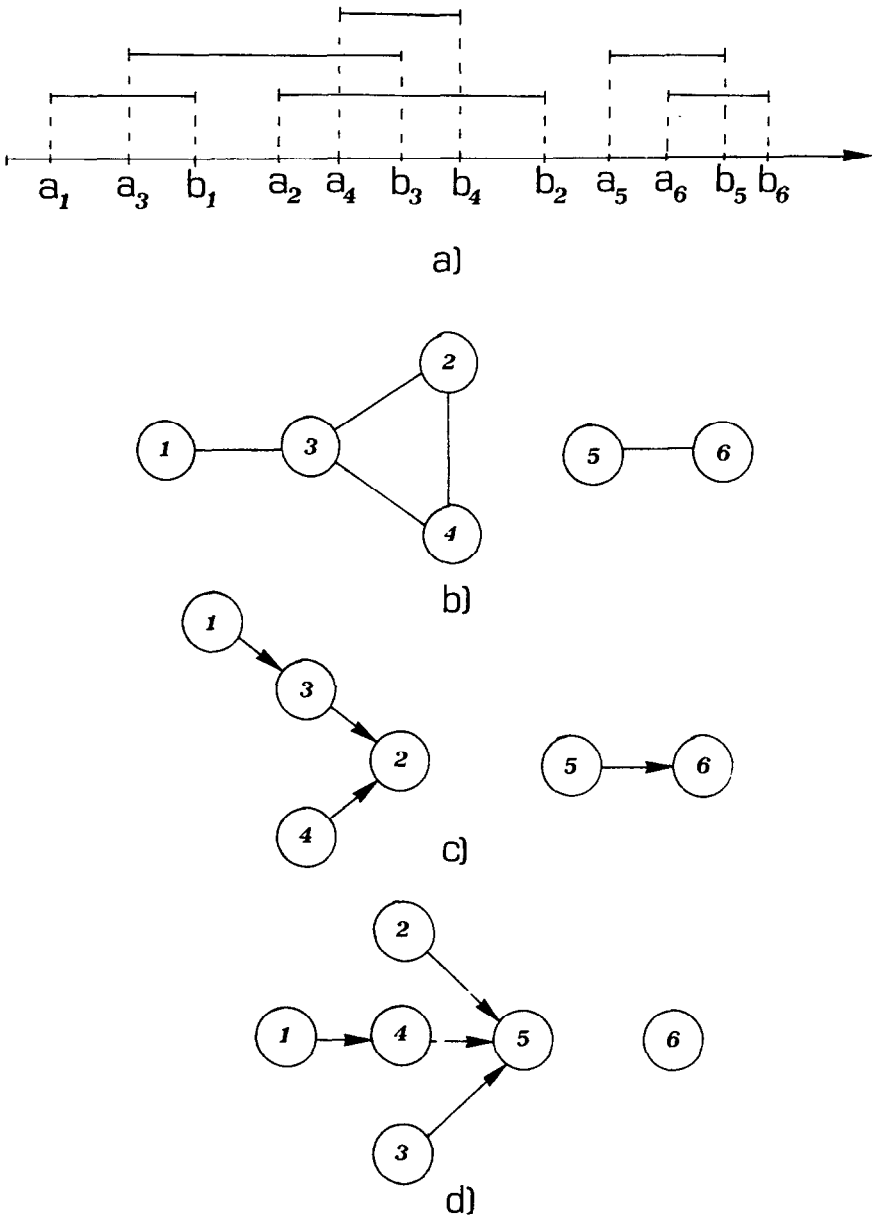

Fig. 1. (a) An interval family; (b) the resulting interval graph; (c) linked lists corresponding to the *right*( ) relation; (d) linked lists corresponding to the *next*( ) relation.

$\phi(j) := 1$
for each $i \neq j$ **do in parallel** $\phi(i) := 0$ **end for**
for $h := 1$ **to** $\lceil \log_2 n \rceil$ **do**
  for each i **such that** $link(i) \neq 0$ **do in parallel**
    if $\phi(i) = 1$ **then** $\phi(link(i)) := 1$ **end if**
    $link(i) := link(link(i))$
  end for
end for

As an example, consider the list (representing a directed leaf-to-root path) shown in Fig. 2. The numbers inside the circles identify the intervals (i.e. nodes of the tree),

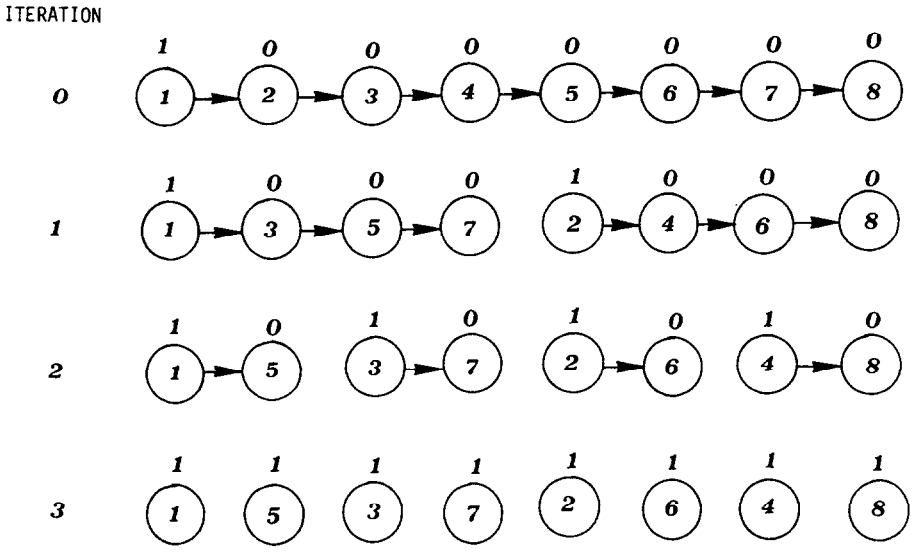

Fig. 2. Transmitting the value 1 within a linked list.

while the numbers outside give the $\phi(\ )$ values. At each iteration, each interval looks two nodes to the right and updates its link, while the value 1 is transmitted to the node on the right, provided it exists.

The parallel time complexity of this algorithm is $O(\log n)$ and requires $O(n)$ processors.

3. Maximum weighted clique

A *clique* of a graph is a set of nodes such that every two nodes in the set are joined by an edge. In an interval graph, a clique corresponds to a set of intervals all intersecting at a common point in $\{a_1, \ldots, a_n\}$. Thus, in order to find a *maximum weighted clique*, it is sufficient to find a point $p \in \{a_1, \ldots, a_n\}$ for which the sum of the weights of the intervals including p is maximum.

First of all, observe that no interval having negative weight can belong to a maximum clique, unless all weights are nonpositive, in which case such a clique consists simply of a single interval having maximum weight. Therefore, we can assume that all the intervals having nonpositive weight have been deleted from I.

We can solve this problem by slightly changing the algorithm given by Dekel and Sahni [5] for solving its maximum cardinality version. We begin by sorting the $2n$ points $a_1, \ldots, a_n, b_1, \ldots, b_n$ in increasing order. This can be done in $O(\log n)$ time on $O(n^2/\log n)$ processors using the sorting algorithm of Muller and Preparata [12]. Let $c_1, \ldots, c_{2n}$ be the resulting sorted sequence. Set $z_j = w_k$, if c_j is an a_k, and $z_j = -w_k$, if c_j is a b_k. Then $r_j = \sum_{i=1}^{j} z_i$ gives the sum of the weights of all intervals containing the point c_j, provided that c_j is an a_k. The r_j's, $1 \leq j \leq 2n$, can be computed using the partial sum algorithm of Dekel and Sahni [6], which takes $O(\log n)$ time and uses $O(n/\log n)$ processors. Once the r_j's have been computed, we find j^*

such that $r_{j^*} = \max\{r_j\}$ and c_{j^*} is an a_k in $O(\log n)$ time using $O(n/\log n)$ processors. In this way, each $i \in I$ belongs to the maximum weighted clique if and only if $a_i \leq c_{j^*} < b_i$. This can be tested in $O(1)$ time using $O(n)$ processors. Therefore, the overall time complexity to find a maximum weighted clique is $O(\log n)$. The number of processors used is $O(n^2/\log n)$. Since this problem can be solved sequentially in $O(n \log n)$ time, the resulting EPU is $\Omega(\log n/n)$.

4. Maximum weighted independent set

An *independent set* of a graph is a set of nodes such that no two nodes in the set are joined by an edge. In an interval graph, such a set corresponds to a set of mutually nonoverlapping intervals.

As in the previous problem, we can assume without loss of generality that all weights are nonnegative. We now show how the maximum weighted independent set problem in an interval graph $G(I)$ can be solved in parallel via a reduction to a *shortest path* problem on an appropriate directed graph H.

First of all, we augment I with two 'dummy' intervals, say 0 and $n+1$, such that $b_0 < a_{first(I)}$ and $b_{last(I)} < a_{n+1}$. These two intervals have zero weight, that is, $w_0 = w_{n+1} = 0$. Then we define H as follows. The nodes of H correspond to the intervals in $I' = I \cup \{0, n+1\}$. There is an arc from node i to node j in H if and only if $b_i < a_j$. The *length* of arc (i, j) is $-w_i$. It is easy to see that a maximum weighted independent set in $G(I')$ corresponds to a shortest path between nodes 0 and $n+1$ in H. Such a path can be computed in parallel in $O(\log^2 n)$ time using $O(n^3/\log n)$ processors [15], once the *node-to-node adjacency matrix* for H is constructed. This can easily be done in $O(1)$ time using $O(n^2)$ processors. A maximum weighted independent set for $G(I)$ can thus be obtained by eliminating nodes 0 and $n+1$ from the shortest path. Thus the overall running time is $O(\log^2 n)$ using $O(n^3/\log n)$ processors. Since an $O(n \log n)$ sequential algorithm is known [11], the resulting EPU is $\Omega(1/n^2)$.

In the unweighted case, a faster and more efficient parallel algorithm can be devised. In this case, a maximum cardinality independent set S may be readily defined in an inductive fashion (e.g. see [7]). Using our notation, we have that $first(I) \in S$; moreover, if $i \in S$, then $next(i) \in S$, too.

Therefore, S can be computed by firstly evaluating $next(i)$ for all $i \in I$. This requires $O(\log n)$ time and $O(n^2/\log n)$ processors. Then we set $link(i) = next(i)$ for all $i \in I$ and $\phi(first(I)) = 1$. Finally, we use the algorithm given in section 2 for transmitting the value 1 through the linked list beginning with $first(I)$. Intuitively, $\phi(i)$ is a characteristic function: $\phi(i) = 1$ whenever $i \in S$, while is equal to zero otherwise. Since the $\phi(\,)$ values can be evaluated in $O(\log n)$ time using $O(n)$ processors, the overall computation takes $O(\log n)$ time and uses $O(n^2/\log n)$ processors. Since the fastest sequential algorithm requires $O(n \log n)$ time, the resulting EPU is $\Omega(\log n/n)$.

5. Minimum clique cover

In the *minimum clique cover* problem, we are asked to find a minimum cardinality partition of the nodes of the graph into cliques. In an interval graph, such a cover can be easily obtained once a maximum cardinality independent set has been found (e.g., see [7]).

In fact, if $S=\{i(1),\dots,i(m)\}$ is a maximum cardinality independent set, then there exists a minimum clique cover consisting of exactly m cliques $C_{i(1)},\dots,C_{i(m)}$ such that each interval $i(j)\in S$ belongs to $C_{i(j)}$, while each $h\notin S$ belongs to the clique $C_{i(k)}$ if and only if

$$b_{i(k)}=\min\{b_i: i\in S \text{ and } a_h<b_i<b_h\}.$$

This condition can be verified for a given h in $O(\log n)$ time using $O(n/\log n)$ processors. It is indeed sufficient to set $y_i=b_i$, if $a_h<b_i<b_h$ and $i\in S$, or $y_i=\infty$ otherwise, and then evaluating i^* (see Section 2). Therefore, we can determine for all $h\notin S$ the appropriate clique to which h belongs in $O(\log n)$ time using $O(n^2/\log n)$ processors, once a maximum cardinality independent set has been found. Since this requires the same time and processors bounds, the overall computation takes $O(\log n)$ time and uses $O(n^2/\log n)$ processors. The resulting EPU is $\Omega(\log n/n)$, because an $O(n\log n)$ time sequential algorithm is known.

6. Minimum weighted dominating set

A *dominating set* of a graph is a subset S of the nodes such that every node of the graph either is in S or is joined by an edge to at least one node in S.

As before, the problem of finding a dominating set having minimum overall weight in an interval graph $G(I)$ can also be solved by reducing it to a shortest path problem on an appropriate directed graph H.

Firstly, let us temporarily assume that there is no negative weight. Again, we consider the augmented interval family I' as defined in Section 4. The nodes of H correspond to the intervals in I'. There is an arc (i,j) in H with length w_i if and only if $j\in P(i)\cup Q(i)$, where:

$$P(i)=\{k: a_i<a_k<b_i<b_k\}, \quad \text{and}$$

$$Q(i)=\{k: a_k>b_i \text{ and there is no } h \text{ with } b_i<a_h<b_h<a_k\}.$$

One may readily verify that any path from node 0 to node $n+1$ in H corresponds to a dominating set for $G(I')$. Indeed, for each (i,j) in such a path, the corresponding intervals $i,j\in I'$ dominate all the intervals k for which $a_i\le a_k<b_j$ or $a_i<b_k\le b_j$. Thus all the intervals appearing in such a path dominate all the intervals in I' (note that intervals 0 and $n+1$ dominate only themselves, since they are isolated). On the other hand, it is not true that any dominating set for $G(I')$ corresponds to a path from node 0 to node $n+1$ in H. However, this is true for any *proper* dominating

set, that is, one corresponding to a proper interval subfamily. Indeed, let $S=\{i(1),\ldots,i(k)\}$ be a proper dominating set. Assume without loss of generality that $a_{i(1)}<\cdots<a_{i(k)}$. Since S is proper, it cannot include any interval properly contained within another interval in S. Therefore, $b_{i(1)}<\cdots<b_{i(k)}$ must result. Observe that arc $(i(j), i(j+1))$ always exists in H. In fact, intervals $i(j)$ and $i(j+1)$ either overlap, in which case $i(j+1)\in P(i(j))$, or do not overlap, in which case $i(j+1)\in Q(i(j))$, since otherwise S could not be a dominating set. Since all weights are nonnegative, any minimum weighted dominating set is also proper. Therefore, it corresponds to a shortest path from node 0 to node $n+1$ in H. The minimum dominating set for $G(I)$ is then obtained from such a path by deleting nodes 0 and $n+1$.

In order to apply the parallel shortest path algorithm [15], the node-to-node adjacency matrix for H has to be computed. This can be done in $O(\log n)$ time using $O(n^2)$ processors. Indeed, a single $P(i)$ can be computed in O(1) time using $O(n)$ processors. To compute a single $Q(i)$, $next(i)$ has to be found in $O(\log n)$ time using $O(n/\log n)$ processors. Then $Q(i)=\{k: b_i<a_k<b_{next(i)}\}$ can be computed in O(1) time using $O(n)$ processors. Therefore, the overall computation to find a minimum weighted dominating set takes $O(\log^2 n)$ time and uses $O(n^3/\log n)$ processors, and is due to the parallel shortest path algorithm [15]. Since an $O(n\log n)$ time sequential algorithm is known [11], the resulting EPU is $\Omega(1/n^2)$.

In the case there are negative weights, we construct H as before, except that the length of arc (i, j) is set to zero whenever $w_i<0$. Once a shortest path has been found, a minimum weighted dominating set for $G(I)$ is obtained by deleting intervals 0 and $n+1$ and by adding all the intervals having negative weights [11]. The time and processor bounds remain the same as before.

Finally, in the unweighted case, a minimum cardinality dominating set can be found by means of a faster and more efficient parallel algorithm. Let us consider the translation into our notation of Booth and Johnson's sequential algorithm for this problem [3]. To do this, we firstly define $right'(i)$ in a slightly different manner than $right(i)$: $right'(i)=j$, if $b_j=\max\{b_k: a_k<b_i\leq b_k\}$, $=0$, if $i=0$. In practice, the condition $b_i<b_k$ has been replaced by $b_i\leq b_k$ and the case for $i=0$ has been added. In this way, $right'(i)=i$ can occur whenever there is no interval $j\neq i$ overlapping with i and ending after i. As usual, $right'(i)$ for all $i\in I$ can be computed in $O(\log n)$ time using $O(n^2/\log n)$ processors. Once this is done, we compute $next(i)$ for all i (using the same amount of time and number of processors). A minimum cardinality dominating set S may be inductively defined as follows [3]: $right'(first(I))\in S$; moreover, if $i\in S$, then $right'(next(i))\in S$ too. Of course, the composition of $right'(\,)$ and $next(\,)$ can be carried out in O(1) time using $O(n)$ processors. Let us denote the resulting pointers as $link(\,)$. Once these pointers are computed, we set $\phi(right'(first(I)))=1$, and use the algorithm given in Section 2 for propagating this value through the resulting linked list, so as to obtain the actual minimum dominating set. This requires $O(\log n)$ time and $O(n)$ processors. Thus the overall computation takes $O(\log n)$ time and uses $O(n^2/\log n)$ processors. Since an $O(\log n)$ time sequential algorithm is known, the resulting EPU is $\Omega(\log n/n)$.

7. Hamiltonian circuit

A *Hamiltonian path* (*circuit*) of a graph is an ordering of the nodes such that every two (cyclically) consecutive nodes of the ordering are joined by an edge.

Let us consider only proper interval graphs. Necessary and sufficient conditions for the existence of a Hamiltonian circuit in a proper interval graph are provided in [2]. In particular, assume there are at least three intervals in I. Then $G(I)$ has a Hamiltonian circuit if and only if the intervals in $I-\{first(I), last(I)\}$ can be partitioned into two disjoint subsets J and K, such that:

(1) $right(first(I)) \in J$; moreover, if $j \in J$, then $right(j) \in J$; finally, $last(I) \in J$, too;

(2) the subgraph having interval family $K \cup \{first(I), last(I)\}$ has a Hamiltonian path.

These conditions lead to a fast parallel algorithm. Firstly, we set $\phi(right(first(I))) = 1$ and propagate this value through the $right(\,)$ list using the algorithm given in Section 2. Once this is over, we test the value of $\phi(last(I))$. If it is 0, then there is no Hamiltonian circuit (since the graph is unconnected [2]); otherwise, we mark all the intervals having $\phi(i) = 1$ but $first(I)$ and $last(I)$. Secondly, we check whether the subgraph having unmarked interval family has a Hamiltonian path. This is a very straightforward task [2]; it is sufficient to sort the unmarked intervals into decreasing order of their leftmost endpoints and check whether every two consecutive intervals in the sorted sequence do overlap or not. If not, there is no Hamiltonian circuit in $G(I)$. Otherwise, the actual Hamiltonian circuit is given by:

(1) all marked intervals sorted into increasing order of their rightmost endpoints, followed by

(2) all unmarked intervals sorted into decreasing order of their leftmost endpoints.

Since the two sorting phases require $O(\log n)$ time using $O(n^2/\log n)$ processors [12], a Hamiltonian circuit of a proper interval graph can also be found (provided it exists) with the same time and processor bounds. The fastest sequential algorithm has an $O(n \log n)$ time complexity [2]. Thus the EPU of our parallel algorithm is $\Omega(\log n/n)$.

8. Minimum bandwidth

A graph $G(N, E)$ has *bandwidth* h if there is a linear ordering of the node set N, i.e., a one-to-one function $f: N \to \{1, \ldots, |N|\}$, such that for each edge $u, v \in E$, $|f(u) - f(v)| \leq h$. The problem of finding a smallest such an h is called *minimum bandwidth problem*. It corresponds to the problem of minimizing the bandwidth of a symmetric matrix by simultaneous row and column permutations (e.g., see [14]).

Assume $G(I)$ is a proper interval graph. The minimum bandwidth of $G(I)$ can be found by indexing the intervals in I by increasing values of their rightmost end-

points and then set $f(i) = i$, namely, choose as f the identity function. This indexing can be carried out in $O(\log n)$ time using $O(n^2/\log n)$ processors.

The minimum value of the bandwidth is equal to $k-1$, where k is the size of a maximum clique for $G(I)$. To check this, assume the intervals in I are indexed as described above. Define an *overlap* clique of I to be a maximal set of intervals all intersecting at a common point in $\{a_1, \ldots, a_n\}$. As proved by Orlin et al. [13], in a proper interval graph whose intervals are indexed by increasing b_i's, each overlap clique C is *consecutive*, namely, $C = \{i, i+1, \ldots, j\}$ for some i and j. Therefore, for any $x, y \in C$, $|f(x) - f(y)| < |C|$. Let C^* be a maximum clique of G having, say, size k. Since all maximal cliques of G are overlap cliques, including thus C^* (see Section 3), and $k-1$ is a lower bound for the minimum bandwidth, the proof follows. Since k can be determined in $O(\log n)$ time using $O(n^2/\log n)$ processors, the minimum bandwidth problem can be solved with the same computational effort and the same number of processors. An $O(n \log n)$ time sequential algorithm based on sorting easily solves this problem. Therefore, the EPU of the above parallel algorithm is $\Omega(\log n/n)$.

9. Further research

We believe that further research can be done in the following main directions. On one hand, the parallel algorithms given here for the maximum weighted independent set and minimum weighted dominating set problems have a quite low EPU. Therefore, one can search for faster and/or more efficient algorithms for these problems. On the other hand, we assumed throughout this paper that an interval family was already known. When this is not the case, one has to determine whether a given graph is an interval graph and construct, in the affirmative case, an interval family for it. This can be done in linear time by means of a sequential algorithm (e.g., see [7]), but no parallel algorithm has been devised up to now.

References

[1] B. Awerbach, A. Israeli and Y. Shiloach, Finding Euler circuits in logarithmic parallel time, Proc. 16th Annual ACM Symp. on Theory of Computing (1984) 249–257.

[2] A.A. Bertossi, Finding Hamiltonian circuits in proper interval graphs, Inform. Proc. Lett. 17 (1983) 97–101.

[3] K. Booth and J.H. Johnson, Dominating sets in chordal graphs, SIAM J. Comput. 11 (1982) 191–199.

[4] E. Dekel, D. Nassimi and S. Sahni, Parallel matrix and graph algorithms, SIAM J. Comput. 10 (1981) 657–675.

[5] E. Dekel and S. Sahni, Parallel scheduling algorithms, Oper. Res. 31 (1983) 24–49.

[6] E. Dekel and S. Sahni, Binary trees and parallel scheduling algorithms, IEEE Trans. Comput. 32 (1983) 307–315.

[7] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs (Academic Press, New York, 1980).

[8] U.I. Gupta, D.T. Lee and J.Y.-T. Leung, Efficient algorithms for interval graphs and circular arc graphs, Networks 12 (1982) 459–467.

[9] R. Karp and A. Widgerson, A fast parallel algorithm for the maximal independent set problem, Proc. 16th Annual ACM Symp. on Theory of Computing (1984) 266–272.

[10] G.A.P. Kindervater and J.K. Lenstra, An introduction to parallelism in combinatorial optimization, Discrete Appl. Math. 14 (1986) 135–156.

[11] G.K. Manacher and C.J. Smith, Efficient algorithms for new problems on interval graphs and interval models, manuscript (1984).

[12] D.E. Muller and F.P. Preparata, Bounds to complexities of networks for sorting and switching, J. ACM 22 (1975) 195–201.

[13] J.B. Orlin, M.A. Bonuccelli and D.P. Bovet, An $O(n^2)$ algorithm for coloring proper circular arc graphs, SIAM J. Algebraic Discrete Methods 2 (1981) 88–93.

[14] C.H. Papadimitriou, The NP-completeness of the bandwidth minimization problem, Computing 16 (1976) 263–270.

[15] M.J. Quinn and N. Deo, Parallel graph algorithms, Comput. Surveys 16 (1984) 319–348.

[16] F.S. Roberts, Discrete Mathematical Models, with Applications to Social, Biological and Environmental Problems (Prentice-Hall, Englewood Cliffs, NJ, 1976).